Haaga-Helia
University of Applied Sciences

Influence of Clean Code on Unit Testing

Tatiana Laneva

Bachelor's Thesis
Degree Programme in BIT
2020

Haaga-Helia
University of Applied Sciences

Abstract

Date

Author(s)
Tatiana Laneva

Degree programme
BIT

Report/thesis title
Influence of Clean Code on Unit Testing

Number of pages and appendix pages
29 + 6

Agile methodology is widely used in today's Software Development. One of the stages of Agile Development is Maintainance. It highly influences the overall cost of Development. Maintainability of the Software can be improved in the Implementation stage by introducing Unit testing. Another way to improve maintainability is by applying principles and practices of Clean code. This parallelism in the aspect of improving maintainability yields further investigation of how clean code principles influence the testing process.

This thesis investigates the effect of applying Clean code principles on Unit testing. One of the key metrics of the quality of Unit testing is code coverage. This research aims to prove the hypothesis that code coverage can be improved by applying Clean code principles. Two questions were answered: 1) How the code coverage of ad hoc code is different from the code coverage of the Clean code?; 2) How well ad hoc evolving written code can be tested?

Case study research method is applied. Two cases are created. For the first case code written in an ad-hoc approach is used, for the second case, Clean code is used. Experienced software developers were asked to write unit tests for these two cases. Code coverages are measured, and the written tests are reviewed for both cases.

It is observed that the Clean code approach achieves higher code coverage compare to ad hoc approach code coverage. Moreover, it is concluded that the existence of the embedded functionality in the ad hoc approach leads to difficulties in the testing process.

Keywords
Clean code, Unit testing, Agile methodology, code coverage

Table of contents

List of figures

List of code snippets

List of tables

1 Introduction

Agile Development is a widely used methodology in today's Software Development. It contains Requirements, Design, Implementation, Verification, and Maintainance stages (Knippers, D., 2011.). Maintenance is crucial for any software. It includes modifications, corrections, and optimisations of code (Erdil, K., Finn, E., Keating, K., Meattle, J., Park, S. and Yoon, D., 2003.).

Maintainability of the Software can be improved in the Implementation stage by introducing Unit testing (Osherove, R. 2014, 10.). Unit testing is a method to test the code in the Implementation stage. Software can be split for different units. Each Unit can be tested by providing inputs and checking the produced output, comparing it with expected values. The quality of Unit testing can be measured by code coverage (Williams, T.W., Mercer, M.R., Mucha, J.P. and Kapur, R., 2001, January.).

Another way to improve maintainability is by applying principles and practices introduced by Robert C. Martin in the "Clean Code A Handbook of Agile Craftsmanship" book. This book is about how to write code, which is easy to alter to meet business demands (Martin, R., 2008, xix.).

Both Unit Testing and Clean code are applied in the Implementation stage to improve the Maintainability of Software. This parallelism yields further investigation of how clean code principles influence the testing process. This research addresses the following questions 1) How the code coverage of ad hoc code is different from the code coverage of the Clean code?; 2) How well ad hoc evolving written code can be tested? This research aims to prove the hypothesis that code coverage can be improved by applying Clean code principles.

The thesis contains four chapters: Theoretical background, Empirical part, and the Discussion part. In the Theoretical background Unit testing, code coverage metrics, and Clean Code principles are detailed. In the Empirical part, the case study research method is applied; study cases descriptions are provided, output documents of the research are included and analysed. Finally, in the discussion part, the findings are concluded, and future works are suggested.

2 Theoretical framework

An important stage of Agile Software Development methodology is Maintenance (Kajko-Mattsson, M., Lewis, G.A., Siracusa, D., Nelson, T., Chapin, N., Heydt, M., Nocks, J. and Snee, H., 2006, September.). To be able to maintain a Software, the Implementation stage should be done according to rules, such as Clean code (Martin, R., 2008, xx.) and Unit Testing (Osherove, R. 2014, 10.). In the theoretical part, the Unit testing is described, metric to measure the quality of unit tests is given, and Clean Code practices are covered.

2.1 Unit testing

Unit testing is a method to check any unit of Software by giving various input data and compare output data with expected values. According to Roy Osherove, there is no single developer who never did Unit testing. Some of using Console to see method output. Some are testing via the user interface. All these methods are not corresponding to criteria, which making Unit Tests valuable for the project (Osherove, R. 2014, 5.).

The complete definition of unit testing is: "A unite test is an automated piece of code that invokes the Unit of work being tested, and then checks some assumptions about a single end result of that Unit. A unit test is almost always written using a unit testing framework. It can be written and run easily; it's trust-worthy, readable, and maintainable. It's consistent its results as long as production code hasn't changed" (Osherove, 2014, 11.).

Roy Osherove provides the criteria of the Unit test, which makes it valuable for the Software. First, unit tests should be written and running fast. Second, the task of any unit test is to check actual functionality and not only pass successfully. A unit test has to be easy to read and understand, easy to change if technical documentation will require these changes in the logic. Finally, a unit test should be stateless. It should always produce the same output for the same input until the production code is not modified (Osherove, R. 2014, 11.).

2.1.1 Example of Unit Tests

```
public double Sum(
    double a, double b)
{
   return a + b;
}
```

```
[TestMethod]
public void Sum_Equal_To_Expect_Value()
{
  Assert.AreEqual(Sum(4,5), 9);
}

[TestMethod]
public void Sum_Not_Equal_To_Expect_Value()
{
  Assert.AreNotEqual(Sum(4, 5), 8);
```

```
}
```

Code snippet 1. Unit test example

Code snippet 1 shows an example of unit tests written for calculating the sum of the inputs. The tests check if the function produces the correct output. The unit test can either pass or fail. The unit test passes if the expected output matches the produced by the unit result. If the unit has been altered, and the production of this unit changed, the test fails (Osherove, R. 2014, 11.).

2.1.2 Purpose of Unit Testing

"Why do most developers fear to make continuous changes to their code? They are afraid they'll break it! Why are they afraid they'll break it? Because they don't have tests." (Robert C. Martin 2011.)

Unit testing is introduced to the code to check the correctness of the implemented logic. By checking the unit output in Unit tests, implements different scenarios, the potential number of bugs can be significantly reduced (Osherove, R. 2014, 11.).

Another essential aspect of Agile Software Development is refactoring. Refactoring is the process of the constant improvement of the code quality by identifying and reducing design issues in the written code. Unit testing is crucial for refactoring. Moreover, refactoring is impossible without unit testing. Written unit tests ensure that improved code behaves the same way as before refactoring (Vonken, F. and Zaidman, A., 2012, October.).

Unit tests reduce costs. When tests are done in the early stages, and for each Unit, it is faster to find the issue and faster to fix the problem (Delgado, D. and Martinez, A., 2013, October.).

2.2 Code coverage

Code coverage is a metric, which indicates the quality of unit testing. It represents how much of the code was run in unit tests (Hollén, J.W. and Zacarias, P.S., 2015). There are plenty of tools available on the market nowadays to visualise the code coverage, i.e., JetBrains' dotCover, OpenCover, NCover, Visual Studio Pro, Ncrunch, etc. As a result, these tools produce a percentage number for code coverage metric (Osherove, R. 2014, 160.).

However, according to R.Osherove, this number will not be thorough enough for committing how tests covered the right code. If the code coverage number is, e.g., 20%, it

is clear that code was not sufficiently covered by tests. Although, the opposite, e.g., 80-90 %, does not a guarantee of excellent code coverage, and the review is needed (Osherove, R. 2014, 160.).

2.3 Clean Code

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." (Martin Fowler, 2008.) Clean code is a methodology that allows developers to produce easy to maintain, easy to understand, and easy to change codebase. Clean code principles come from mistakes and failures of developers who realise that the approach "Get things done" is not the right in the software development lifecycle (Robert C. Martin 2011, 2.).

Applying Clean code principles in the Implementation stage of the Agile workflow will ensure that maintaining the product goes effortlessly. In the following parts, the principles of the Clean code paradigm are covered. They are used in the Code samples part in the Clean code example. All of the Clean code principles are written by Robert Martin in his book "Clean Code A Handbook of Agile Craftsmanship" (Robert C. Martin 2011.).

2.3.1 Meaningful names

One of the first rules of the Clean code paradigm is meaningful names. It applies to variables, functions, arguments, classes, and packages. Names should reveal the intent of the created instance to improve the readability of the code. For example, the variable name "h" does not indicate what this variable is made for and what information it stores. Moreover, "hours" for the variable name could tell that this variable represents in hours, but the question hours of what is missing. At the same time, the variable name "employeeAbsenceInHours" tells that this instance stores the number of hours when the employee was absent (Robert C. Martin 2011, 17.).

Names should be pronounceable, searchable and should not disinform the reader. Names should be understandable for developers who read the code in the future. Developers should not show how smart or funny they are by using "DestroyMyUniverse" instead of "Delete" for the function name. Standard mathematical notation or names of design patterns should be applied for naming. For example, the State pattern class should contain "State" in its name. A variable name should be a noun; a function name should be a verb. "And" word should be avoided in names. If there is "and" in the name, splitting the instance should be considered (Robert C. Martin 2011, 17.).

There are a set of simple rules, which should be applied for naming. All of them claim that name should be understandable, easy to read, and descriptive. The right name improves the readability of the code. Choosing the right name takes time, but it saves much more when the instance is used (Robert C. Martin 2011, 17.).

2.3.2 Functions

A critical aspect of writing clean functions is the Single Responsibility Principle. It means that one function should do one and only one thing. The function should either act (void functions), either answer for a question, but never both. As well, creating side effects in the function should be avoided (Robert C. Martin 2011, 31.).

Functions should be written in the way it is not needed to check the signature or to write comments. It should be clear what function is doing in the calling line. These can be achieved by creating a descriptive name for a function and by taking care of function's arguments. As was mentioned in the previous chapter, the function name should contain a verb and a noun. This name should precisely describe what this function is doing. The number of arguments should be as less as possible, the ideal is 0, and the maximum is 3. If it is impossible to achieve, creating an argument object should be considered (Robert C. Martin 2011, 31.).

Functions should be short—ideally, 3-4 lines per one function. Within the function can be only one level of abstraction. It means the scope in the function should always be at the same level. Switch statements should be avoided, and replaced with abstract factories. The good practice is to throw exceptions rather than returning errors. As well don't repeat yourself principle (DRY) should be followed. The function should not have any repeatable code (Robert C. Martin 2011, 31.).

2.3.3 Comments

During the Maintainance stage, the code base is continuously being evolved. There is no such thing as a static code. In one week, month or year, function, class, or any other instance of the code is modified. That is why it is dangerous to write comments. Eventually, all comments in the code are going to be deprecated. Comment, which was left for explaining what the function is doing, may stay, but the function itself most likely is modified. Thus disinformation in the code might appear. That's why comments should be avoided as much as possible (Robert C. Martin 2011, 53.).

Furthermore, most of the time, comments are written to cover the failure in producing clean, understandable, and easy to follow code. Code snippet 2 shows the example of the code with a comment. The reason this comment appeared is an insufficient function naming. On the opposite, Code snippet 3 shows the same function with an explicit name. It is evident, from Code snippet 3 there is no need to write a comment. The better practice to express what this code does is by applying meaningful naming (Robert C. Martin 2011, 53.).

```
//Function to convert elements to string format
public IEnumerable<string> GetData(IList<double> data)
{
  return data.Select(element => element.ToString());
}
```

Code snippet 2. Comment

```
public IEnumerable<string> ConvertElementsToString(IList<double> data)
{
  return data.Select(element => element.ToString());
}
```

Code snippet 3. Meaningful name

Another reason why comments can appear in the code base is the commented out code. During the refactoring or Implementation, the developer can decide that code should be re-written. Still, the original one is left in the file because it might be needed in the future. Commenting out the old code is a bad practice. All commented out code should be removed to make a class easy to read and to remove distraction. Source control always helps in case of this code is needed in the future. Source control is the practice, which widely used nowadays. It allows to track all changes of all team members was made in the codebase (Robert C. Martin 2011, 53.).

To conclude, comments should be avoided. Although, there are specific use cases when comments can appear: 1) Legal comments about copyright and authorship, if it is required; 2) Comments for particular domain names or logic, which involve decisions beyond the Implementation; 3) TODO comments (Robert C. Martin 2011, 53.).

2.3.4 Error Handling

The way how the errors can be handled is an essential part of clean code. Error handling should not depend on the main logic. There should be strict rules, what to consider as an error in the Software. For example, file input/output can throw an error, and that error should be handled. On the other hand, if there are no elements in the list, and developer

tries to address to the first element (getting null argument exception), it should be considered as a part of the logic. Throwing exception for this case is inappropriate (Robert C. Martin 2011, 103.).

Returning errors instead of throwing exceptions is a bad practice. Errors should not be passed further. Exceptions should contain all needed information, such as message, parameter, etc.. Another part of error handling is Nulls. Null should not be passed for the functions, and should not be returned. Instead of returning Null, an empty instance should be considered. If Null is passed to the function, an exception should be thrown. Try, catch, and finally block should be written if it is possible to get exceptions in the program workflow (Robert C. Martin 2011, 103.).

2.3.5 Classes

According to the Clean Code rules, a class should be small. A small class is a class with one and only one responsibility. It means that any class should be possible to describe without conjunctions (and, but, if and or). The Single Responsibility Principle (SRP) should be applied to classes. A class should be only one reason to change, only one responsibility handled, and class should collaborate with other classes to achieve the desired behaviour of the System. A class should contain a small number of attributes and should have high cohesion. The class name should describe the purpose of its creation (Robert C. Martin 2011, 135.).

2.3.6 Test-Driven Development

Test-Driven Development (TDD) is the approach of writing unit tests before code. TDD is a part of Clean code philosophy. TDD allows developers to think about all edge cases and the logic which is needed to be implemented before the actual code is writing. This method is recommended as a highly efficient way of Agile Development cycle. Main advantages of TDD are good code coverage, clean and understandable code. It takes a lot of discipline to start to practice TDD (Robert C. Martin 2011, 121.).

3 Empirical part

The goal of the Empirical part is to address the following research questions: 1) How the code coverage of ad hoc code is different from the code coverage of the Clean code?; 2) How well ad hoc evolving written code can be tested?

To answer these questions, the case study research method is used. Subchapter 3.1 describes the chosen method and provides arguments, why the method is suitable for this research; subchapter 3.2 describes the workflow for this research; subchapter 3.3 provides code samples for the case study; subchapter 3.4 collects the data for the analysis; subchapter 3.5 analyse the collected data. In the final subchapter 3.6, findings are summarized.

3.1 Case study research method

The research aims to prove the following hypothesis: code coverage can be improved by applying Clean code principles. The first step of prooving this theory is to define research questions: 1) How the code coverage of ad hoc code is different from the code coverage of the Clean code?; 2) How well ad hoc evolving written code can be tested? These questions can be answered by comparing code coverages of Clean code written code and ad hoc written code. The output type of the code coverage is a percentage. It represents how much of the code was run in unit tests, which is quantitative. At the same time, as mentioned in the theory part, it is not enough to compare code coverage percentages. The more detailed review is needed to take on the unit tests itself, to prove that code is tested well. The output data of this research needs to be qualitative and quantitative.

According to Scot A. Miller, there are five main methodologies to collect qualitative data: narrative, phenomenology, ethnography, grounded theory and case study. Narrative, phenomenology and ethnography research methodologies are collecting live, personal or group experiences; the grounded theory is creating theory based on the research. The case study analyses single or multiple cases. (Miller, S.A., 2017.).

Since any personal experience is considered as subjective, narrative, phenomenology and ethnography methodologies can not provide an objective comparison of the code coverages. Moreover, it is impossible to calculate the code coverage based only on the experience of interviewees. Therefore, Grounded theory methodology is not suitable for this research because the hypothesis already defined.

A case study is a research method which allows to retrieve and analyse the dynamic data from the static input. A case study can be applied to prove a theory, test a theory or to

provide a description. The case study can contain several cases, where each of the cases is some static input. Interviews, questionaries, observations and archives are typical data collection methods for a case study research. The output results can be qualitative or quantitative (Eisenhardt, K.M., 1989.).

The case study is considered as the most suitable research method for this thesis. For the static input, two code samples are created: the first one is an ad hoc written code, and the second is a code following Clean code principles. The dynamic output data is the Unit tests written by interviewees for the static input data. The dynamic output data is analyzed by calculating the code coverage for both approaches. Unit tests are reviewed for further analysis.

3.2 Research process

Four experienced software engineers were asked to write unit tests to two code samples (ad hoc code approach and Clean Code approach code). Developer 1 has 7+ years of experience in Software Development; developer 2 has 2+ years of software development experience; developer 3 has 8+ years of Software Development experience; developer 4 has 12+ years of experience in Software Development.

The description of code samples is given in chapter 3.3: Code samples. The first case is to ask developers to tests the ad hoc code approach implementation of the task specification. The ad hoc code can be found in Appendix 1: Code snippet 12. The second case is to ask developers to test the Clean Code approach implementation of the task specification. The Clean Code can be found in Appendix 2: Code snippets 13-18.

Developers received the code samples via a GitHub link and were asked to commit their unit tests in separate branches. The code coverage is calculated for each of the branches separately by using JetBrains Re-Sharper (Jetbrains.com/resharper 2020). Unit tests review is performed to evaluate the correctness of the code coverage.

3.3 Code samples

In the theoretical framework, the subchapter 2.3 Clean code describes the basic rules of writing readable and maintainable code. Based on these rules, two code samples were created. One of the code samples is following clean code principles (Clean code approach). Another one represents an ad hoc approach, where clean code principles are omitted (Ad hoc approach). The subchapter: 3.3.1 Task Specification contains the description of code samples. In the subchapters: 3.3.2 Ad hoc code Example and 3.3.3 Clean Code Example codes are introduced.

3.3.1 Task Specification

Ad hoc and Clean code approaches are developed to solve one problem: calculate the perimeter of a triangle in 3-dimensional space. The input parameters are three points A, B, and C with three coordinates X, Y, and Z. The output is the calculated value, which represents the perimeter of a triangle. The following formula can calculate the perimeter: AB + BC + AC = Perimeter, where AB, BC, and AC are distances between points A and B, B and C, A, and B respectively.

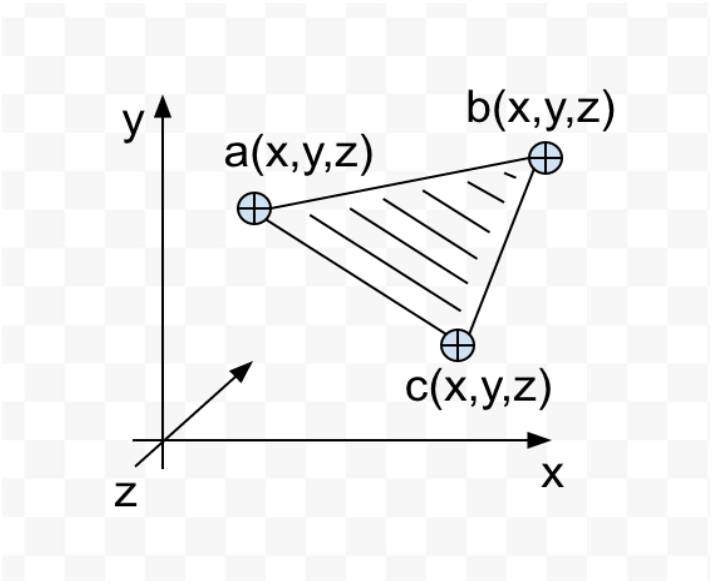

Figure 1. Triangle in 3-dimensional space

To perform this calculation, first, the possibility to build the triangle from the three points need to be checked. The points should be in different locations in space. It means that coordinates X, Y, and Z of all three points should not be equal. Moreover, points should not be collinear. It means that vectors AB and AC should not be parallel to each other.

The program should accept three points, check if they are valid to perform the calculation and return the result. The result is the perimeter of the triangle.

3.3.2 Ad hoc code Example

The ad hoc example can be found in Appendix 1, code snippet 12. For this example, clean code principles are omitted and the program developed in an ad hoc approach. As a result, ad hoc example contains 1) absence of meaningful names for class, function, and variables; 2) long and only one function to handle all logic; 3) comments, which are covering poorly readable code; 4) returning nulls instead of proper error handling and, finally, 5) the class violate the Single responsibility principles.

```
var t1 = double.NaN;
var t2 = double.NaN;
var t3 = double.NaN;
```
Code snippet 4. Insufficient naming

Code snippet 4 is a part of ad hoc code and demonstrates the insufficient naming. It is hard to understand what variables t1, t2, and t3 are stood for. To modify this part of code time needs to be spent to realise that t1 t2 and t3 are coefficients of two variables. The absence of meaningful names can be spotted in the Perimeter class in ad hoc code.

```
//calculate the distance between points and return their sum
// as the answer
//calculate distance between point1 and point2
var d1 = Math.Sqrt(Math.Pow(point2.Item1 - point1.Item1, 2) +
                   Math.Pow(point2.Item2 - point1.Item2, 2) +
                   Math.Pow(point2.Item3 - point1.Item3, 2));
//calculate distance between point2 and point3
var d2 = Math.Sqrt(Math.Pow(point3.Item1 - point2.Item1, 2) +
                   Math.Pow(point3.Item2 - point2.Item2, 2) +
                   Math.Pow(point3.Item3 - point2.Item3, 2));
//calculate distance between point 3 and point 1
var d3 = Math.Sqrt(Math.Pow(point1.Item1 - point3.Item1, 2) +
                   Math.Pow(point1.Item2 - point3.Item2, 2) +
                   Math.Pow(point1.Item3 - point3.Item3, 2));

return d1 + d2 + d3;
```
Code snippet 5. Comments

The next issue with the Perimeter class can be observed in Code snippet 5. Instead of providing meaningful names and moving logic to separated functions, comments were added. Comments do not improve readability, but a developer has to jump between comments and code to understand the meaning of these lines.

```
/*check if vector AB parallels to vector AC. If AB parallel to AC, then
points A, B and C will be collinear and triangle will be not possible
to build. */
var t1 = double.NaN;
var t2 = double.NaN;
var t3 = double.NaN;
if (ac.Item1 - 0 > 0.001)
{
  t1 = ab.Item1 / ac.Item1;
}

if (ac.Item2 - 0 > 0.001)
{
  t2 = ab.Item2 / ac.Item2;
}

if (ac.Item3 - 0 > 0.001)
{
  t3 = ab.Item3 / ac.Item3;
```

```
}

if (Math.Abs(t1 - t2) < 0.001 && Math.Abs(t1 - t3) < 0.001)
  return null;
```

Code snippet 6. Avoiding function separation

Code snippet 4 and Code snippet 6 illustrate replacing function separation by comments. Single Responsibility principle was violated by creating only one function, which is responsible for all calculations and validations. This function can not be described without using words, "and", "if", "or" and "but".

```
//check if point1 equal to point2
if (Math.Abs(point1.Item1 - point2.Item1) < 0.001 &&
    Math.Abs(point1.Item2 - point2.Item2) < 0.001 &&
    Math.Abs(point1.Item3 - point2.Item3) < 0.001) return null;
//check if point2 equal to point3
if (Math.Abs(point2.Item1 - point3.Item1) < 0.001 &&
    Math.Abs(point2.Item2 - point3.Item2) < 0.001 &&
    Math.Abs(point2.Item3 - point3.Item3) < 0.001) return null;
//check if point1 equal to point3
if (Math.Abs(point1.Item1 - point3.Item1) < 0.001 &&
    Math.Abs(point1.Item2 - point3.Item2) < 0.001 &&
    Math.Abs(point1.Item3 - point3.Item3) < 0.001) return null;
```

Code snippet 7. Inappropriate error handling

Inappropriate error handling is shown in Code snippet 7. Instead of throwing exceptions if input data did not pass through the validation process, the CalculatePerimeter function returning Null. In that case, it is unclear what kind of error appeared and why input data is invalid.

The most crucial issue with the Perimeter class is a violation of the Single Responsibility Principle. Class Perimeter is responsible for the validation of input parameters and calculation of the final results. In other words, this class can not be described without using the word "and", which means this class should be divided at least to two parts.

3.3.3 Clean Code Example

The Clean code example can be found in Appendix 2, code snippets 13-18. This example is created by applying Clean Code principles, which are described in the Theoretical framework, subchapter 2.3. Four different classes are created: Vector, Point3D, TriangleValidator, and TrianglePerimeterCalculator. As well, two interfaces are implemented: ITriangleValidator and IVector. Each of the classes is following single responsibility principles, and each of the classes has a meaningful name.

```
public class TrianglePerimeterCalculator
{
  private readonly ITriangleValidator _triangleValidator;

  public TrianglePerimeterCalculator(ITriangleValidator
                          triangleValidator)
  {
    _triangleValidator = triangleValidator;
  }

  public double Calculate(Point3D point1, Point3D point2,
                                              Point3D point3)
  {
    _triangleValidator.CheckIfTriangleCanBeBuilt(point1, point2,
                                                    point3);
    return CalculateTrianglePerimeter(point1, point2, point3);
  }

  private double CalculateTrianglePerimeter(Point3D point1, Point3D
                                             point2, Point3D point3)
  {
    return point1.GetDistanceToPoint(point2) +
           point1.GetDistanceToPoint(point3) +
           point2.GetDistanceToPoint(point3);
  }
}
```

Code snippet 8. TrianglePerimeterCalculator class

No functions in the clean code approach solution violate the Single responsibility principle. All of the functions have three or four lines in the body. The purpose of the functions is described by their name, and not by comments above or inside function declaration. In the Code snippet 8, an example of the functions written for TrianglePerimeterCalculator is shown.

```
 private void CheckPointsForCollinearity()
 {
   AssignVectors();
   // If vector AB parallel to AC, then points A, B and C are collinear
   CheckIfVectorsParallel();
 }
```

Code snippet 9. Example of the comment in the Clean code

In the Code Snippet 9, a comment is shown. This comment is placed in the TriangleValidator class to explain the domain logic, which might not be in the scope of developer knowledge. The amount of the comments is minimised in the classes. Instead, meaningful namings for functions, classes, and variables are implemented.

```
 private void CheckPointsForEquality()
 {
   if (_point1.Equals(_point2) || _point1.Equals(_point3) ||
       _point2.Equals(_point3))
   {
```

```
      throw new ArgumentException(EqualPointsError,
                                  nameof(TriangleValidator));
    }
  }

  private void CheckIfVectorsParallel()
  {
    if (_vectorPoint1Point2.IsParallelToVector(_vectorPoint1Point3))
    {
      throw new ArgumentException(CollinearPointsError,
                                    nameof(TriangleValidator));
    }
  }
```

Code snippet 10. Error handling

Error handling is done differently, compare with the ad hoc approach. There are no null returns, but the class TriangleValidator is created. When the triangle is impossible to build with the given points, the TriangleValidator class throws exceptions. In the Code snippet 10, the exception handling is shown. Introducing exceptions requires to change the way of TriangleCalculator call from the Program class. In the Code snippet 11, the try-catch block is demonstrated.

```
private static void PrintCleanCodeResults()
{
  var perimeterCalculator = new TrianglePerimeterCalculator(
                                      new TriangleValidator());
  try
  {
    var result = perimeterCalculator.Calculate(new Point3D(1, 1, 1),
                        new Point3D(1, 1, 1), new Point3D(3, 3, 3));
    Console.WriteLine(result);
  }
  catch (Exception exception)
  {
    Console.WriteLine(exception.Message);
  }
}
```

Code snippet 11. Try - catch block

3.4 Review of Code coverage

In the subchapters, 3.2 Ad hoc code example, and 3.3 Clean code example, the code samples are described. These code samples are published in the GitHub repository <https://github.com/TaniaLaneva/Thesis-code-samples>. Four developers involved in writing unit tests for ad hoc code and clean code. They submitted their Unit tests in dedicated branches. Each branch is cloned to the local machine, and code coverage is calculated by

the Re-Sharper tool (Jetbrains.com/resharper 2020). Re-Sharper tool has a unit test coverage calculation, which is providing code coverage. This chapter contains the Unit Test Coverage and unit tests overview of the ad hoc and Clean code approaches.

3.4.1 Calculated code coverages

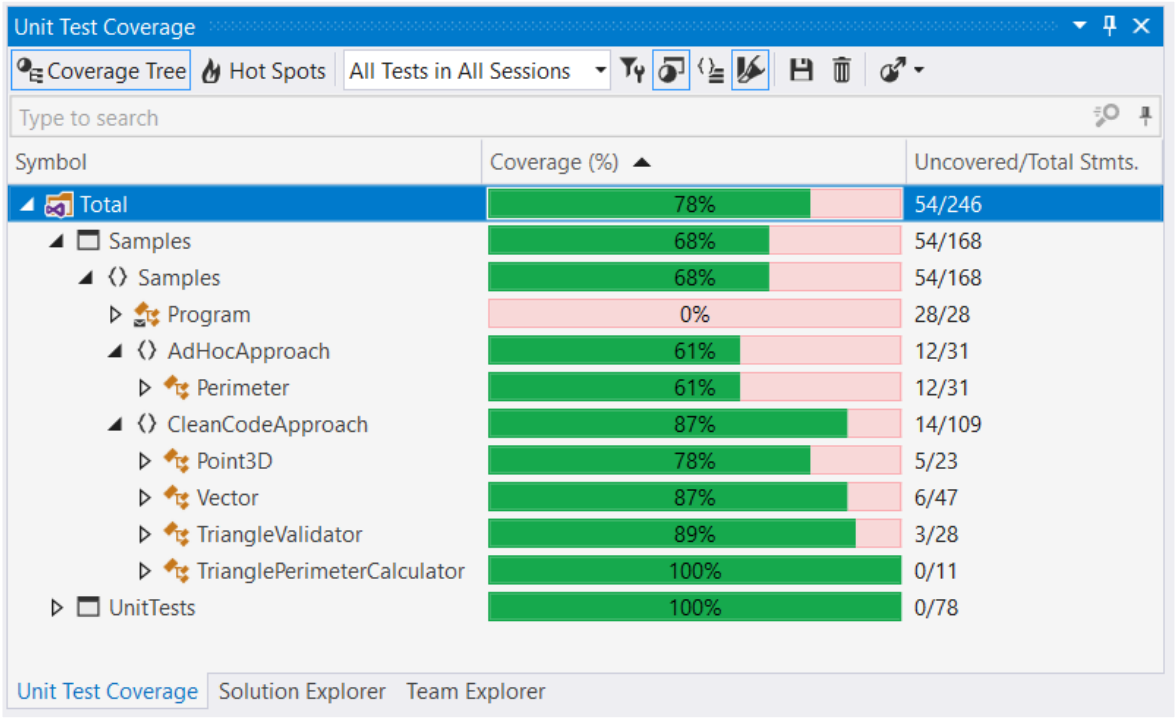

Figure 2. Calculated code coverage for Developer 1.

Figure 2 shows the calculated code coverage for Developer 1. It demonstrates the auto-generated code coverage for AdHocApproach class (Perimeter) and CleanCodeApproach classes (Point3D, Vector, TriangleValdator, and TrianglePerimeterCalculator). Figure 2 shows that Developer 1 managed to produce Unit tests for both cases: Clean code approach and ad hoc approach. Code coverage for the ad hoc code sample, which contains only Perimeter class, is 61%, and for Clean code sample in aggregate is 87%, which includes: Point3D class 78%, Vector class 87%, TriangleValidator class 89% and TrianglePerimeterCalculator class 100%.

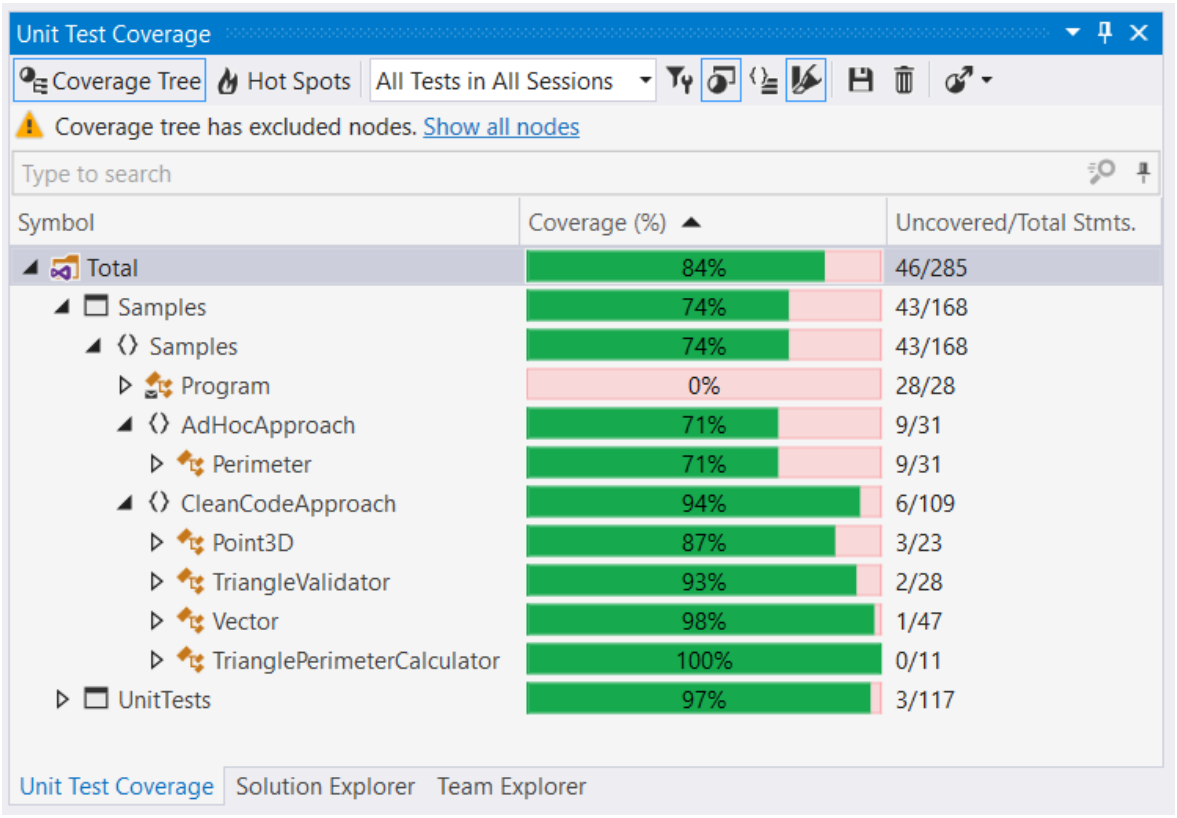

Figure 3. Calculated code coverage for Developer 2.

Figure 3 shows the calculated code coverage for Developer 2. It demonstrates the auto-generated code coverage for AdHocApproach class (Perimeter) and CleanCodeApproach classes (Point3D, Vector, TriangleValdator, and TrianglePerimeterCalculator). Figure 3 shows that Developer 2 manages to produce Unit tests for both cases: Clean code approach and ad hoc approach. Code coverage for the ad hoc code sample, which contains only Perimeter class, is 71%, and for Clean code sample in aggregate is 94%, which includes: Point3D class 87%, Vector class 93%, TriangleValidator class 98% and TrianglePerimeterCalculator class 100%.

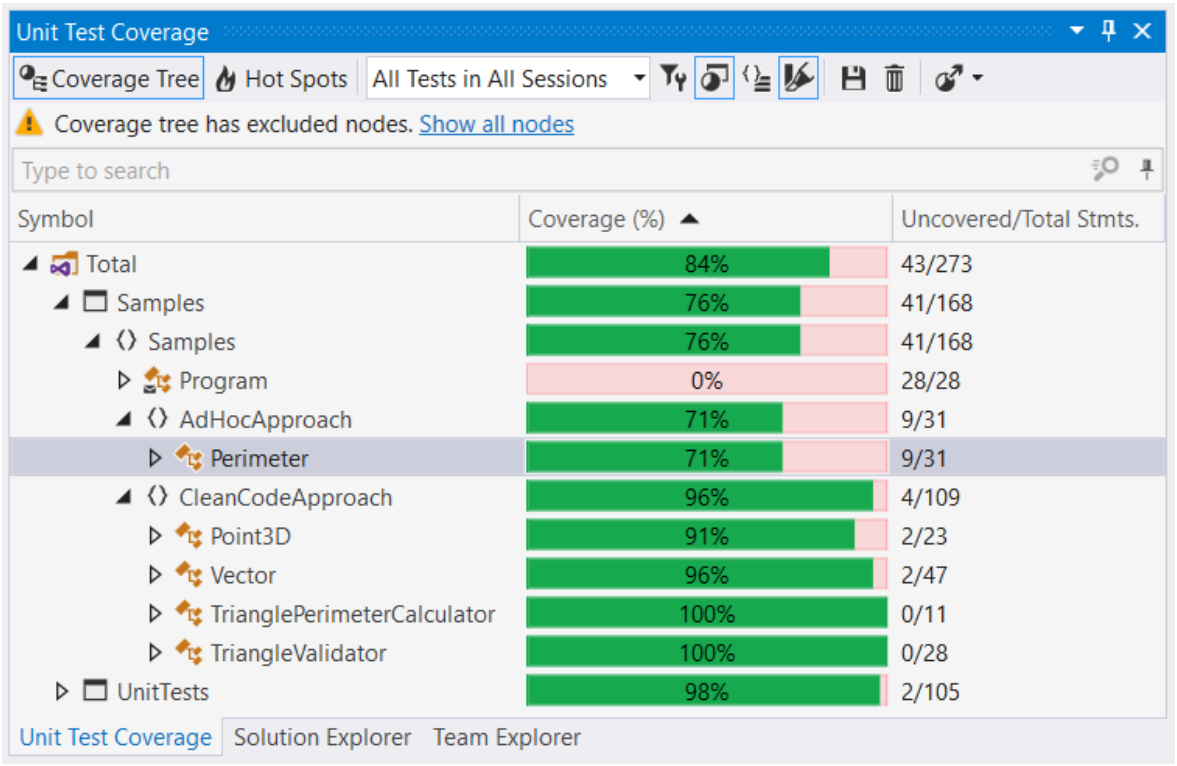

Figure 4. Calculated code coverage for Developer 3.

Figure 4 shows the calculated code coverage for Developer 3. It demonstrates the auto-generated code coverage for AdHocApproach class (Perimeter) and CleanCodeApproach classes (Point3D, Vector, TriangleValdator, and TrianglePerimeterCalculator). Figure 4 shows that Developer 3 manages to produce Unit tests for both cases: Clean code approach and ad hoc approach. Code coverage for the ad hoc code sample, which contains only Perimeter class, is 71%, and for Clean code sample in aggregate is 96%, which includes: Point3D class 91%, Vector class 96%, TriangleValidator class 100% and TrianglePerimeterCalculator class 100%.

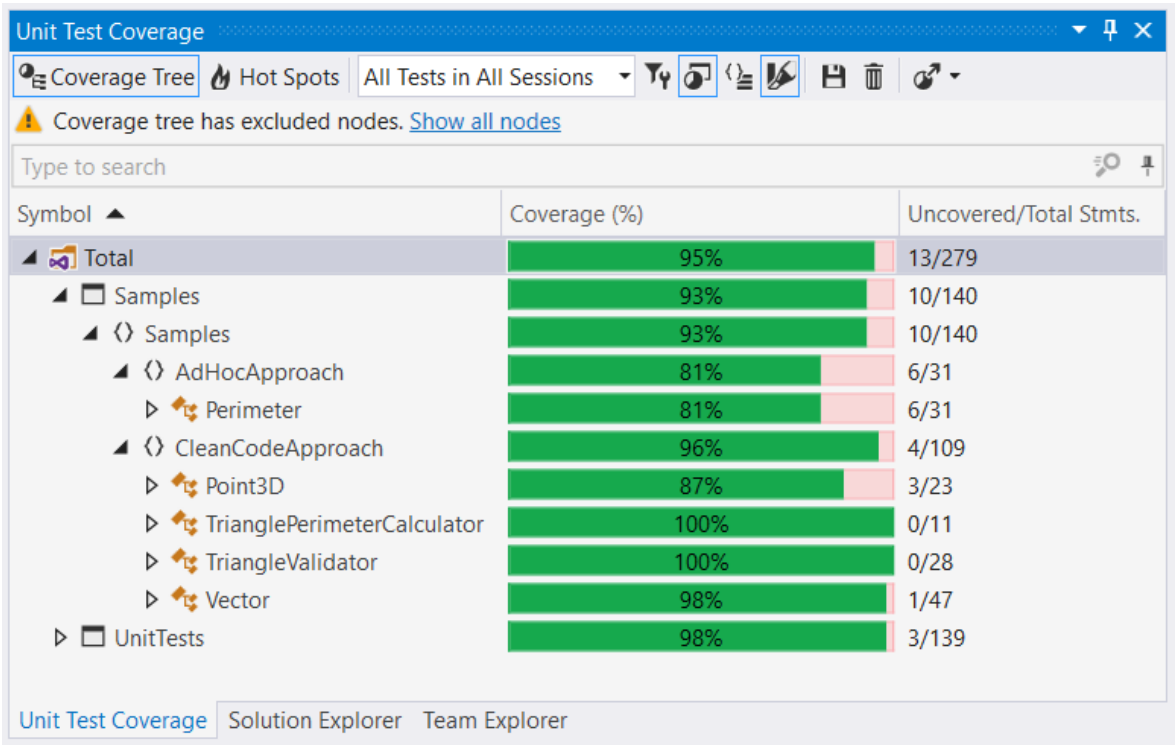

Figure 5. Calculated code coverage for Developer 4.

Figure 5 shows the calculated code coverage for Developer 4. It demonstrates the auto-generated code coverage for AdHocApproach class (Perimeter) and CleanCodeApproach classes (Point3D, Vector, TriangleValdator, and TrianglePerimeterCalculator). Figure 5 shows that Developer 4 manages to produce Unit tests for both cases: Clean code approach and ad hoc approach. Code coverage for the ad hoc code sample, which contains only Perimeter class, is 81% and for clean code sample code coverage in aggregate is 96%. Clean code sample includes: Point3D class 87%, Vector class 98%, TriangleValidator class 100% and TrianglePerimeterCalculator class 100%.

3.4.2 Unit tests overview of ad hoc approach

In this chapter, the review of Unit tests for ad hoc approach is performed for each developer. The ad hoc approach contains one class – Perimeter. In subchapter 3.2 Ad hoc code Example, it is claimed that Perimeter class violates the Single Responsibility Principle. The perimeter class is responsible for:

- Validation of points
  - Checking points for the equality
  - Building vector
  - Validation vectors are not parallel
- Calculation distance between two points
- Calculation of the perimeter

Developer 1 tested the following responsibilities of the Perimeter class:

- Calculation of the perimeter
    - One test with valid input coordinates
- Validation of points. Checking points for the equality
    - One test. Attempt to calculate the perimeter with points, two out of three where are equal.

Developer 2 tested the following responsibilities of the Perimeter class:

- Calculation of the perimeter
    - One test with valid coordinates
- Validation of points. Checking points for equality
    - One test. Attempt to calculate the perimeter with equal points
- Validation of points. Validation vectors are not parallel
    - One test. Attempt to calculate the perimeter with collinear points

Developer 3 tested the following responsibilities of the Perimeter class:

- Calculation of the perimeter
    - Check perimeter calculation for positive points
    - Check perimeter calculation for points from different quarters
- Validation of points. Checking points for the equality
    - One test. Attempt to calculate the perimeter with points, two out of three where are equal

Developer 4 tested the following responsibilities of the Perimeter class:

- Calculation of the perimeter
    - Calculate the perimeter of the equilateral triangle with positive points
    - Calculate the perimeter of the equilateral triangle with negative points
    - Calculate the perimeter with a maximum value
    - Calculate the perimeter with valid random values
- Validation of points. Checking points for the equality
    - Attempt to calculate the perimeter with equal points
    - Attempt to calculate the perimeter with a close to each other points
- Validation of points. Validation vectors are not parallel
    - Attempt to calculate the perimeter with a collinear points

3.4.3 Unit tests overview of Clean code approach

In this chapter, the review of Unit tests for Clean code approach is performed for each developer. Clean code approach contains four classes: Points3D, Vector, TriangleValidator and TrianglePerimeterCalculator. In chapter 3.3 Clean code Example, it is claimed that classes do not violate the Single Responsibility Principle. Each of the classes has a set of public methods, which can be tested.

- Points3D class public functionalities are:
    - Create the point
    - Get the distance between points
    - Operator overloading
    - Equals method override
    - GetHashCode method override

- Vector class public function is to check if two vectors are parallel
- TriangleValidator public function is to check if the triangle can be built with the given points.
- TrianglePerimeterCalculator public function is to calculate the perimeter of a triangle

Developer 1 tested the following functionality:

- Point3D class
  - Check the distance between:
    - Equals points
    - Points in different locations
- Vector class: Check if vectors are parallel:
  - Parallel vectors
  - Not parallel vectors
- TriangleValidator class: Check if a triangle can be built with:
  - Equal points
  - Parallel vectors
- TrianglePerimeterCalculator class
  - Check if TriangleValidator has been called
  - Check the output with valid input points

Developer 2 tested the following functionality:

- Point3D class
  - Check the distance between:
    - Equals points
    - Points in different locations
  - Check the operator overloading
    - Chech values for each coordinate
  - Check equals method override
    - Check unequal points
    - Check equal points
    - Check attempt to compare point with another object
    - Check that point is not equal to null
- Vector class: Check if vectors are parallel:
  - Parallel vectors
  - Not parallel vectors
  - Check vectors with zeros
  - Check zero vectors
  - Check if vector parallel to zero vector
- TriangleValidator class: Check if a triangle can be built with:
  - Equal points
  - Not equal points
  - Parallel vectors
  - Not parallel vectors
- TrianglePerimeterCalculator class
  - Check if TriangleValidator has been called
  - Check the output with valid input points

Developer 3 tested the following functionality:

- Point3D class

        - Point creation
        - Check the distance between:
            - Points in different locations
        - Check the operator overloading
            - Chech values for each coordinate
        - Check equals method override
            - Check equal points
        - GetHashCode method override
            - Check hashes are matching for equal points
- Vector class: Check if vectors are parallel:
    - Parallel vectors
- TriangleValidator class: Check if a triangle can be built with:
    - Equal points
    - Parallel vectors
    - Not equal or parallel points
- TrianglePerimeterCalculator class
    - Check the output with valid input points

Developer 4 tested the following functionality:

- Point3D class
    - Check the distance between:
        - Points in different locations
    - Check equals method override
        - Check equal points
        - Check unequal points
- Vector class: Check if vectors are parallel:
    - Parallel vectors
    - Converging vectors
- TriangleValidator class: Check if a triangle can be built with:
    - Uncollinear points
    - Equal points
    - Colinear points
    - Colinear points with one negative coordinate
- TrianglePerimeterCalculator class
    - Check the output with valid input points
    - Check the output with valid negative points
    - Check the output with a maximum value points

3.5 Data analysis

This research addresses the following questions 1) How the code coverage of ad hoc code is different from the code coverage of the Clean code?; 2) How well ad hoc evolving written code can be tested? The case study research method was used to answer these questions. Experienced developers, who work a substantial amount of years in the field, were asked to test code written in two different approaches: ad hoc code and clean code. By analysing their Unit tests with Re-Sarper tool(Jetbrains.com/resharper 2020) and reviewing them, the following answers are concluded.

Table 1. Code coverage

	Ad hoc approach test coverage	*Clean Code approach test coverage*
Developer 1	61%	87%
Developer 2	71%	94%
Developer 3	71%	96%
Developer 4	81%	96%

Table 1 illustrates the code coverage values for ad hoc and Clean code approaches. The results are calculated by automatic tool Re-Sharper(Jetbrains.com/resharper 2020). It is noticeable that each developer has a higher code coverage percentage with the Clean code approach. Table 2 illustrates the code coverage difference between these two approaches per each developer.

Table 2. Clean code and ad hoc comparison

	Difference between Clean code and ad hoc approaches
Developer 1	26%
Developer 2	23%
Developer 3	25%
Developer 4	15%

Based on Table 2, it can be concluded that the clean code approach improves code coverage compared to the ad hoc approach on average by 22.25%. As noted in the Theoretical Framework, none of the automatic tools provides full information about code coverage. Review of the results needs to be performed. Due to the fact, that code is not the same for both cases; a mapping table (Table 3) is created to link functionalities between Clean code and ad hoc approaches.

Table 3. Functionality mapping

Ad hoc functionality	**Clean code functionality**
-	Create Point
Calculate distance	Get distance between points
Building vector	Operator “-” overloading
-	Equals method override
-	Hashcode override
-	Check if vectors are parallel
1) Checking points for the equality 2) Validation vectors are not parallel	Check if a triangle can be built

Calculation of the perimeter	Calculation of the perimeter

Table 4 ad hoc tests overview represents, what functionalities have been tested by developers for the ad hoc approach.

Table 4. Ad hoc tests overview

		Developer 1	Developer 2	Developer 3	Developer 4
Create Point		Not testable	Not testable	Not testable	Not testable
Get distance between points		-	-	-	-
Operator “-” overloading (building vectors)		-	-	-	-
Equals method override (Checking points for equality)		Not testable	Not testable	Not testable	Not testable
Hashcode override		Not testable	Not testable	Not testable	Not testable
Check if vectors are parallel		Not testable	Not testable	Not testable	Not testable
Check if a triangle can be built:	When points are equal	+	+	+	+
	When vectors are parallel	-	+	-	+
Calculation of the perimeter		+	+	+	+

The ad hoc approach contains embedded operations, such as checking points for equality or checking if vectors are not parallel. These functionalities are not performed in separated units. To test if a triangle can be built with a certain input data is not the same, as checking if the input data is valid. Developers can test if a triangle can be built with parallel vectors, but the important parts, such as checking if vectors are parallel are hardly reachable in the code. Based on this logic, some parts of the functionality in Table 4 are marked as “Not testable”. Leaving such an important and not trivial part like checking vectors is a potential risk that the task was not implemented correctly. It is a potential bug in the program.

Table 5 Clean code tests overview represents, what functionalities have been tested by developers for Clean code approach.

Table 5. Clean code tests overview

		Developer 1	Developer 2	Developer 3	Developer 4
Create Point		-	-	+	-
Get distance between points		+	+	+	+
Operator "-" overloading (building vectors)		-	+	+	-
Equals method override (Checking points for Equality)		+	+	+	+
Hashcode override		-	-	+	-
Check if vectors are parallel		+	+	+	+
Check if a triangle can be built:	When points are equal	+	+	+	+
	When vectors are parallel	+	+	+	+
Calculation of the perimeter		+	+	+	+

Based on the subchapter 3.4.3 Code coverage overview of Clean code approach, it can be obtained that all developers wrote valid tests. Table 5 shows what was tested by each developer. Re-Sharper tool (Jetbrains.com/resharper 2020) is used to calculate the code coverage for Clean code approach. The average of code coverage is 93,25%. After the validation of written tests, it can be inferred that this number is fair to represent the code coverage of the Clean code example.

3.6 Results

Subchapter 3.5 Data analysis leads to the following observations:

1) The calculated code coverage for clean code approach is 22.25% higher on average compared to the ad hoc approach. The average code coverage for clean code approach is 93,25%, for ad hoc approach is 71% (Table 1, Table 2)

2) The calculated code coverage for clean code approach is higher for all developers (Table 1, Table 2)

3) Unit tests review approved the validity of the written tests (Table 4, Table 5)

4) The ad hoc approach contains embedded not testable operations (Table 3, Table 4)

The first research question is: **How the code coverage of ad hoc code is different from the code coverage of the Clean code?** Observations 1 and 2 points out, that code coverage of clean code approach is higher than code coverage of the ad hoc approach. Observation 3 points out the tests, written by developers, are valid. Based on these observations, it can be concluded that by applying the clean code approach, the higher code coverage can be achieved.

Observation 4 leads to the conclusion that auto-generated calculated code coverage metrics do not highlight all the details of how well the code was covered. Moreover, this observation allows answering the second research question: **How well is ad hoc evolving written code can be tested?** Even though ad hoc coverage is 71% average for all developers and all developers wrote valid tests, the existence of the embedded functionality in the ad hoc approach leads to difficulties in the testing process. To conclude, ad hoc evolving written code can not be tested as well as clean code.

4 Discussion

Robert Martin, in his book "Clean code", shares an example of one Software which was entered the market in the late 80s. It has a high demand, but it has to be shut down. The problem was maintaining. At some point, developers just could not implement new features or fix bugs because the codebase becomes a "mess". Clean code principles have been created by Robert Martin to avoid the situation when it's more cost-efficient to start from scratch, rather than continue to maintain the product (Martin, R., 2008, 3.).

In practice, Maintainance stage of Agile Software Development includes Implementation of the new features and fixing bugs. Henning Grimeland Koller, in his Master thesis "Effects of Clean Code on Understandability: An Experiment and Analysis", concludes that Clean code principles improve the time needed to change functionality. On the other hand, Henning Grimeland's research concludes, Clean code does not improve bug fixing time (Koller, H.G., 2016.).

According to Roy Osherove if the code contains a bug, which needed to be fixed, the first thing to do is to write a Unit test. The better code coverage is in the project, the easier and faster it is possible to spot the bug in the System. In other words, bug fixing time and process, in general, can be improved by better code coverage (Osherove, R., 2014, 175.).

This research aims to prove the hypothesis that code coverage can be improved by applying Clean code principles. Two questions were answered: 1) How the code coverage of ad hoc code is different from the code coverage of the Clean code?; 2) How well ad hoc evolving written code can be tested?

Case study research method is applied. Experienced software developers were asked to write unit tests for two cases: Clean code approach code and ad hoc approach code. After that, code coverages for each approach were calculated and compared. Furthermore, unit tests were reviewed to check the validity of the written tests. It has been concluded that the Clean code approach code coverage is higher compared to ad hoc approach code coverage. Moreover, the existence of the embedded functionality in the ad hoc approach leads to difficulties in the testing process.

Good code coverage of Software reduces the number of bugs in the code (Osherove, R., 2014.). The fewer bugs presented in the code base, the easier the Maintainability of Software will be, and the less cost it will require (Erdil, K., Finn, E., Keating, K., Meattle, J., Park, S. and Yoon, D., 2003.). This research proved, that code coverage of the Clean code is higher, rather than code coverage of ad hoc code. Moreover, embedded not

testable functionality can be avoided by following Clean code principles. As a consequence of applying Clean code principles cost of the Maintainability of Software could be reduced.

Clean code principles and Unit testing were never controversial practices. Moreover, one of the most important rules of the Clean code is Test Driven Development (TDD). Writing Unit tests before the actual Implementation forcing a developer to follow the Clean code rules unconsciously (Martin, R., 2008, 121.).

For future researches, more advanced topics, such a Concurrency and Unit Testing of Multithreading systems could be viewed in the Clean code approach in a Unit testing perspective. Clean code principles improve code coverage of the System, although this is not the only advantage of this paradigm. Implementation of the new features process can be viewed and compared between Clean code approach and ad hoc approach in many different aspects.

References

Knippers, D., 2011, June. Agile Software Development and Maintainability. In 15th Twente Student Conf.

Erdil, K., Finn, E., Keating, K., Meattle, J., Park, S. and Yoon, D., 2003. Software maintenance as part of the software life cycle. Comp180: Software Engineering Project, pp.1-49.

Williams, T.W., Mercer, M.R., Mucha, J.P. and Kapur, R., 2001, January. Code coverage, what does it mean in terms of quality?. In Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No. 01CH37179) (pp. 420-424). IEEE.

Osherove, R., 2014. The Art of Unit Testing. 2nd ed. Manning Publications Co. NY

Martin, R., 2008. Clean Code A Handbook of Agile Craftsmanship. 1st ed. Pearson Education, Inc. Boston

Kajko-Mattsson, M., Lewis, G.A., Siracusa, D., Nelson, T., Chapin, N., Heydt, M., Nocks, J. and Snee, H., 2006, September. Long-term life cycle impact of agile methodologies. In 2006 22nd IEEE International Conference on Software Maintenance (pp. 422-425). IEEE.

Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

Vonken, F. and Zaidman, A., 2012, October. Refactoring with unit testing: A match made in heaven?. In 2012 19th Working Conference on Reverse Engineering (pp. 29-38). IEEE.

Delgado, D. and Martinez, A., 2013, October. Cost effectiveness of unit testing: A case study in a financial institution. In 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 340-347). IEEE.

Hollén, J.W. and Zacarias, P.S., 2015. Exploring Code Coverage in Software Testing and its Correlation with Software Quality; A Systematic Literature Review.

Resharper, Jetbrain, computer software, viewed 19 April 2020, <https://www.jetbrains.com/resharper/>

Koller, H.G., 2016. Effects of Clean Code on Understandability: An Experiment and Analysis (Master's thesis).
Miller, S.A., 2017. Developmental research methods. Sage publications.

Eisenhardt, K.M., 1989. Building theories from case study research. Academy of management review, 14(4), pp.532-550.

Unknown Author. Available at <https://math.stackexchange.com/questions/1904614/spreading-points-over-a-triangle-plane-in-3d-space> (Accessed 2 May 2020).

Appendices

Appendix 1. Class to be tested. Ad hoc approach.

```
using System;

namespace Samples.AdHocApproach
{
  //class to calculate perimeter of triangle
  public class Perimeter
  {
    public double? CalculatePerimeter(
                        Tuple<double, double, double> point1,
                        Tuple<double, double, double> point2,
                        Tuple<double, double, double> point3)
    {
      //return null if triangle can not be built from these points.
      //check if point1 equal to point2
      if (Math.Abs(point1.Item1 - point2.Item1) < 0.001 &&
          Math.Abs(point1.Item2 - point2.Item2) < 0.001 &&
          Math.Abs(point1.Item3 - point2.Item3) < 0.001) return null;
      //check if point2 equal to point3
      if (Math.Abs(point2.Item1 - point3.Item1) < 0.001 &&
          Math.Abs(point2.Item2 - point3.Item2) < 0.001 &&
          Math.Abs(point2.Item3 - point3.Item3) < 0.001) return null;
      //check if point1 equal to point3
      if (Math.Abs(point1.Item1 - point3.Item1) < 0.001 &&
          Math.Abs(point1.Item2 - point3.Item2) < 0.001 &&
          Math.Abs(point1.Item3 - point3.Item3) < 0.001) return null;

      //build vector AB and and AC
      var ab = new Tuple<double, double, double>(
                                  point2.Item1 - point1.Item1,
                                  point2.Item2 - point1.Item2,
                                  point2.Item3 - point1.Item3);
      var ac = new Tuple<double, double, double>(
                                  point3.Item1 - point1.Item1,
                                  point3.Item2 - point1.Item2,
                                  point3.Item3 - point1.Item3);

      /*check if vector AB parallel to vector AC. If AB parallel to AC,
       then points A, B and C will be
       collinear and triangle will be not possible to build. */
      var t1 = double.NaN;
      var t2 = double.NaN;
      var t3 = double.NaN;
      if (ac.Item1 - 0 > 0.001)
      {
        t1 = ab.Item1 / ac.Item1;
      }

      if (ac.Item2 - 0 > 0.001)
      {
        t2 = ab.Item2 / ac.Item2;
      }

      if (ac.Item3 - 0 > 0.001)
      {
        t3 = ab.Item3 / ac.Item3;
      }
```

```
      if (Math.Abs(t1 - t2) < 0.001 && Math.Abs(t1 - t3) < 0.001)
          return null;

      //calculate the distance between points and return their sum as
      //the answer
      //calculate distance between point1 and point2
      var d1 = Math.Sqrt(Math.Pow(point2.Item1 - point1.Item1, 2) +
                         Math.Pow(point2.Item2 - point1.Item2, 2) +
                         Math.Pow(point2.Item3 - point1.Item3, 2));
      //calculate distance between point2 and point3
      var d2 = Math.Sqrt(Math.Pow(point3.Item1 - point2.Item1, 2) +
                         Math.Pow(point3.Item2 - point2.Item2, 2) +
                         Math.Pow(point3.Item3 - point2.Item3, 2));
      //calculate distance between point 3 and point 1
      var d3 = Math.Sqrt(Math.Pow(point1.Item1 - point3.Item1, 2) +
                         Math.Pow(point1.Item2 - point3.Item2, 2) +
                         Math.Pow(point1.Item3 - point3.Item3, 2));

      return d1 + d2 + d3;
    }
  }
}
```

Code snippet 12. Ad hoc approach

Appendix 2. Classes to be tested. Clean code approach.

```
namespace Samples.CleanCodeApproach
{
  public class TrianglePerimeterCalculator
  {
    private readonly ITriangleValidator _triangleValidator;

    public TrianglePerimeterCalculator(ITriangleValidator
                                          triangleValidator)
    {
      _triangleValidator = triangleValidator;
    }

    public double Calculate(Point3D point1, Point3D point2,
                                           Point3D point3)
    {
      _triangleValidator.CheckIfTriangleCanBeBuilt(point1, point2,
                                                   point3);
      return CalculateTrianglePerimeter(point1, point2, point3);
    }

    private double CalculateTrianglePerimeter(Point3D point1,
                                     Point3D point2, Point3D point3)
    {
      return point1.GetDistanceToPoint(point2) +
             point1.GetDistanceToPoint(point3) +
             point2.GetDistanceToPoint(point3);
    }
  }
}
```

Code snippet 13. Clean code approach. TrianglePerimeterCalculator class

```
namespace Samples.CleanCodeApproach
{
  public interface ITriangleValidator
  {
    void CheckIfTriangleCanBeBuilt(Point3D point1, Point3D point2,
                                                   Point3D point3);
  }
}
```

Code snippet 14. Clean code approach. TriangleValidator interface

```
using System;

namespace Samples.CleanCodeApproach
{
  public class TriangleValidator : ITriangleValidator
  {
    private Point3D _point1;
    private Point3D _point2;
    private Point3D _point3;
    private Vector _vectorPoint1Point2;
    private Vector _vectorPoint1Point3;
    private const string EqualPointsError = "Can not use same points to
                                  calculate triangle perimeter";
    private const string CollinearPointsError = "Can not use collinear
                              points to calculate triangle perimeter";

    public void CheckIfTriangleCanBeBuilt(Point3D point1,
                                        Point3D point2, Point3D point3)
    {
      AssignPoints(point1, point2, point3);
      CheckPointsForEquality();
      CheckPointsForCollinearity();
    }

    private void CheckPointsForEquality()
    {
      if (_point1.Equals(_point2) || _point1.Equals(_point3) ||
                                     _point2.Equals(_point3))
      {
        throw new ArgumentException(EqualPointsError,
                                      nameof(TriangleValidator));
      }
    }

    private void CheckPointsForCollinearity()
    {
      AssignVectors();
      // If vector AB parallel to AC, then points A, B and C are
      //collinear
      CheckIfVectorsParallel();
    }

    private void CheckIfVectorsParallel()
    {
      if (_vectorPoint1Point2.IsParallelToVector(_vectorPoint1Point3))
      {
        throw new ArgumentException(CollinearPointsError,
                          nameof(TriangleValidator));
      }
    }
```

```
    private void AssignPoints(Point3D point1, Point3D point2,
                                              Point3D point3)
    {
      _point1 = point1;
      _point2 = point2;
      _point3 = point3;
    }

    private void AssignVectors()
    {
      _vectorPoint1Point2 = new Vector(_point2 - _point1);
      _vectorPoint1Point3 = new Vector(_point3 - _point1);
    }
  }
}
```

Code snippet 15. Clean code approach. TriangleValidator class

```
using System;

namespace Samples.CleanCodeApproach
{
  public class Point3D
  {
    public double X { get; }
    public double Y { get; }
    public double Z { get; }

    public Point3D(double x, double y, double z)
    {
      X = x;
      Y = y;
      Z = z;
    }

    public double GetDistanceToPoint(Point3D point)
    {
      return Math.Sqrt(Math.Pow(point.X - X, 2) +
                       Math.Pow(point.Y - Y, 2) +
                       Math.Pow(point.Z - Z, 2));
    }

    public static Point3D operator -(Point3D point1, Point3D point2) =>
      new Point3D(point1.X - point2.X,
                  point1.Y - point2.Y,
                  point1.Z - point2.Z);

    public override bool Equals(object? obj)
    {
      if (obj == null) return false;
      if (!(obj is Point3D point)) return false;
      return X.Equals(point.X) && Y.Equals(point.Y) &&
                                  Z.Equals(point.Z);
    }

    public override int GetHashCode()
    {
      return HashCode.Combine(X, Y, Z);
    }
  }
```

```
}
```

Code snippet 16. Clean code approach. Point3D class

```
namespace Samples.CleanCodeApproach
{
  public interface IVector
  {
    bool IsParallelToVector(Vector vector);
  }
}
```

Code snippet 17. Clean code approach. Vector interface

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Samples.CleanCodeApproach
{
  public class Vector : IVector
  {
    public double X { get; }
    public double Y { get; }
    public double Z { get; }

    public Vector(Point3D point)
    {
      X = point.X;
      Y = point.Y;
      Z = point.Z;
    }

    public bool IsParallelToVector(Vector vector)
    {
      if (!AreZeroValuesMatchingForVector(vector)) return false;
      var listOfCoefficients = GetComponentsCoefficients(
                    ConvertToList(this), ConvertToList(vector));
      return listOfCoefficients.Distinct().ToList().Count == 1;
    }

    private bool AreZeroValuesMatchingForVector(Vector vector)
    {
      if (!AreZeroValuesMatching(vector.X, X)) return false;
      if (!AreZeroValuesMatching(vector.Y, Y)) return false;
      if (!AreZeroValuesMatching(vector.Z, Z)) return false;

      return true;
    }

    private bool AreZeroValuesMatching(double a, double b)
    {
      if (IsEqualToZero(a) && !IsEqualToZero(b)) return false;
      if (!IsEqualToZero(a) && IsEqualToZero(b)) return false;
      return true;
    }

    private IEnumerable<double> GetComponentsCoefficients(
                      IReadOnlyList<double> numeratorValues,
                      IReadOnlyList<double> denominatorValues)
    {
```

```
      for (var i = 0; i < Math.Min(numeratorValues.Count,
                                   denominatorValues.Count); i++)
      {
        if (IsEqualToZero(denominatorValues[i])) continue;
        yield return numeratorValues[i] / denominatorValues[i];
      }
    }

    private List<double> ConvertToList(Vector vector)
    {
      return new List<double> {vector.X, vector.Y, vector.Z};
    }

    private bool IsEqualToZero(double value)
    {
      return value - 0 < 0.001;
    }
  }
}
```

Code snippet 18. Clean code approach. Vector class

Appendix 3. GitHub link

Link to the GitHub with study cases and returned tests from the developers <https://github.com/TaniaLaneva/Thesis-code-samples>